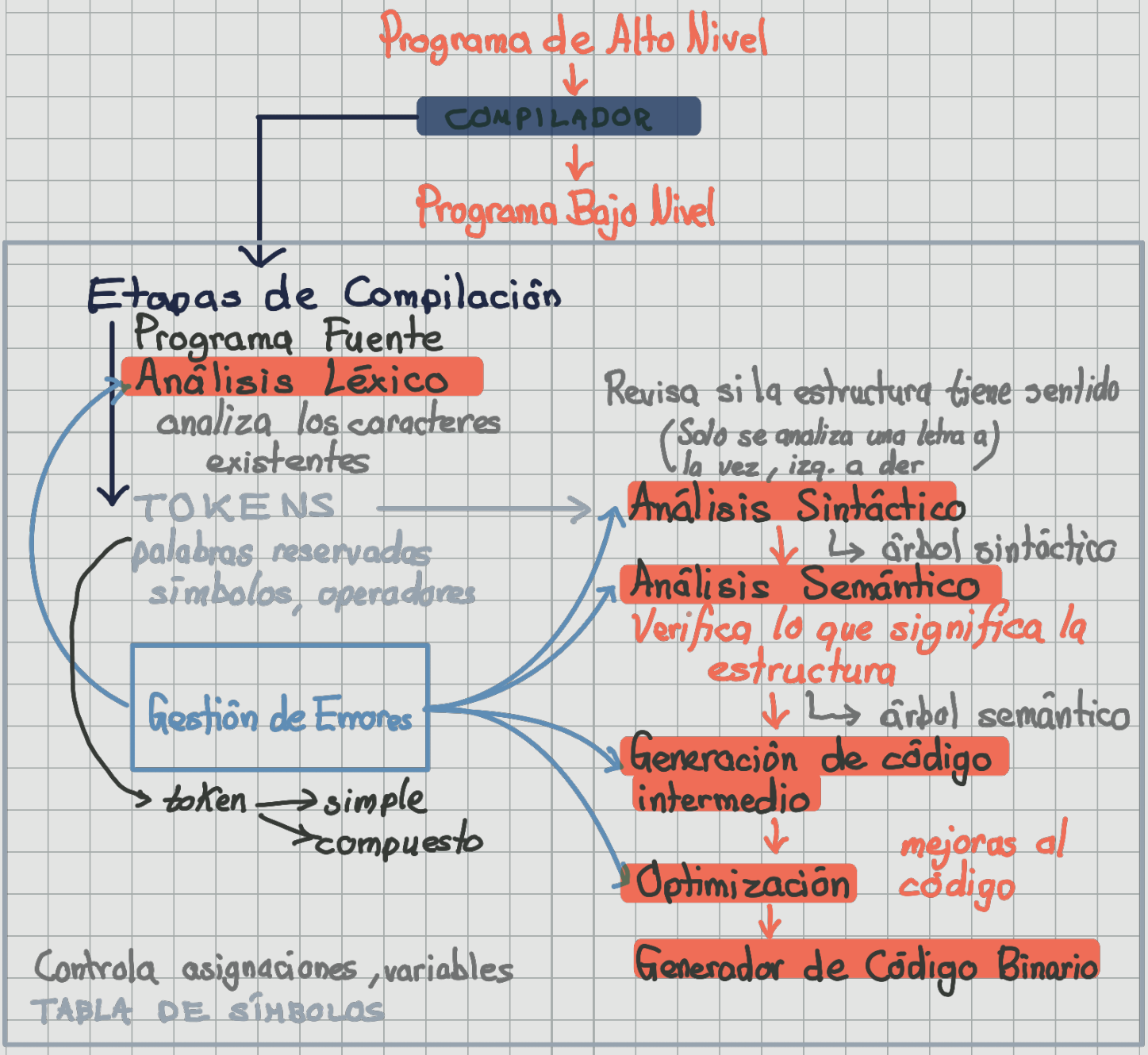
Introducción

Introducción

06/02

¿Qué es un compilador? Traductor de lenguaje de alto a bajo nivel



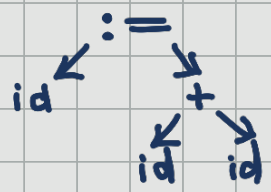
Ej.

posición:= inicial+ velocidad

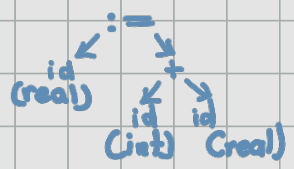
TOKENS

id := id + id

1. Análisis Léxico
   identificador
   (debe ser claro y ententificable)
2. Analizador Sintáctico
   genera el árbol sintáctico o de ejecución
   hace un recorrido der → izq y down → up



3. Analizador Semántico
   identificación de características
   genera árbol semántico



4. Código Intermedio
5. Optimizador
6. Código Binario

Analizador Léxico

Lectura del programa fuente, se recolectan y agrupan los símbolos en unidades significativas llamadas **tokens.**

Secuencia de símbolos = secuencia de tokens

Token

Se divide en:

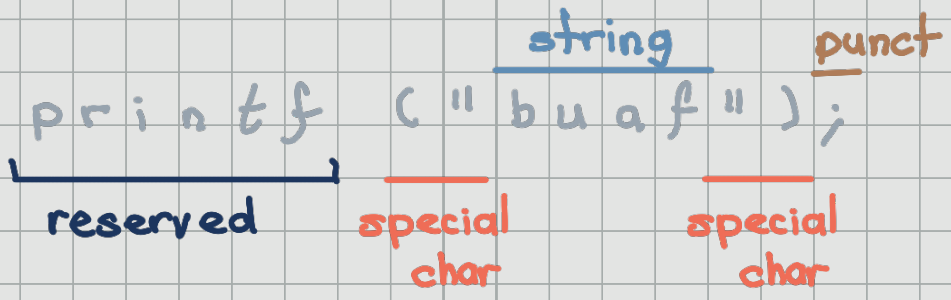
token = {tipo, valor}

Es la unidad básica de un compilador, como una palabra o un signo de puntuación.

printf (" buaf ");

↓ reserved ↓ spec. char

token = {reserved, printf}



token = { reserved w., printf }
token = { special char, (}
token = { special char, " }
token = { string, buaf }
token = { special char " }
token = { special char,) }
token = { punctuation, ; }

11|02

" Muchos piensan equivocadamente, que las circunstancias les determinan, eso no es así... Mas bien lo descubren"

Wayne Dyer

Ej. Sea

```
#include <stdio.h>
void ejemplo(int a, int c)
int main(){
    char eeny[20];
    printf("\n\t buaf: ");
    scanf("%s" &eeny);
...
```

Clasifique en tokens.

token = { especial, # }
= { reserved, include }

= { especial, < }
= { reserved, stdio }
= { puntuación, . }
= { alphabet , h }
= { especial, > }
= { reserved, void }
= { id, ejemplo }
= { puntuación, (}
= { reserved, int }
. . .
= { literal , 20 } ← aplica para números
. . .
= { alphabet, n }
. . .
= { string , buaf }

Ll se considera posteriormente en el análisis gramatical

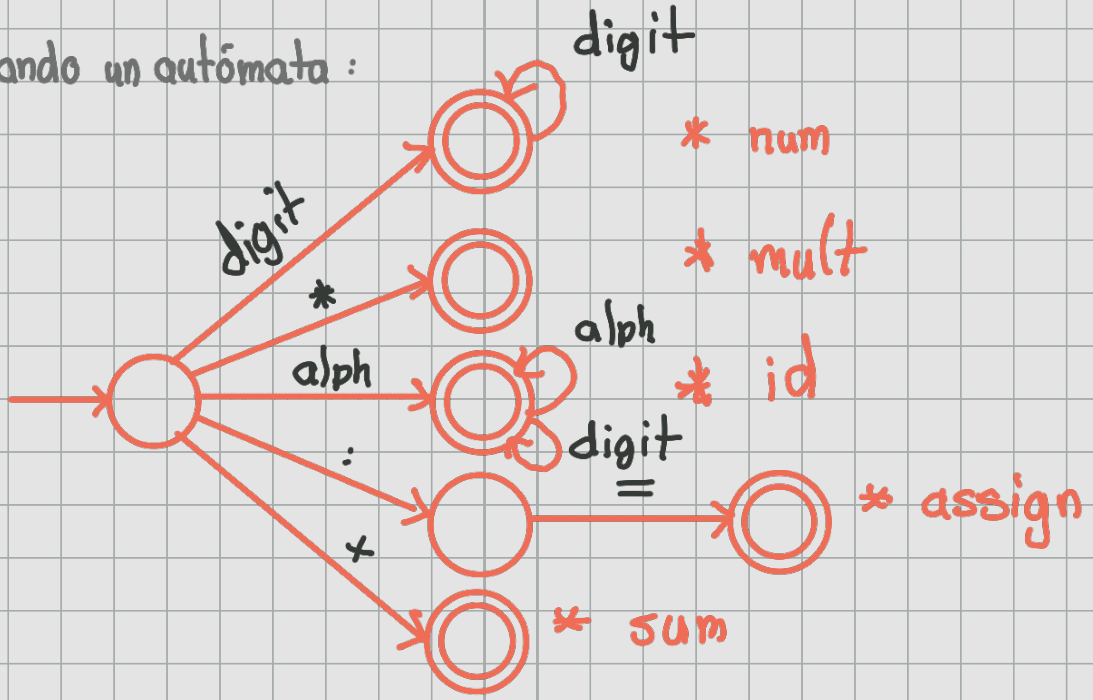
La clasificación de una categoría es sujeto a diseño.

Sin embargo, no es bueno alejarse del canon.

Ej. Identifique y clasifique en tokens:

NUEVO := VIEJO + RAZON * 2025

usando un autómata:



Funciones del Analizador Léxico

Eliminar espacios en blanco, comentarios, saltos de línea, tabuladores...

Agrupar en "paquetes" símbolos
Leer símbolos
Clasificar tokens

Clasificación de Tokens

Específicos

Nunca cambian de forma

operadores {+, -, *, / ...}
reserved words {int, sqrt, void ...}

No específicos

Cambian de forma

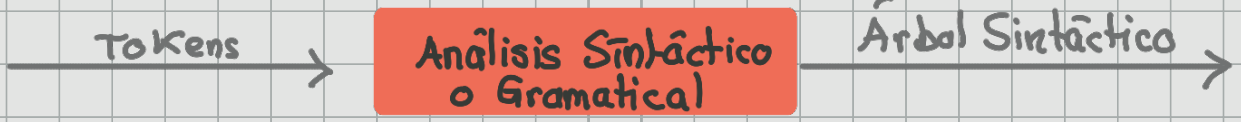
identificadores {Buaf, eepy_medios, mimi2, ...}
literales {números}

"El buen amor es ida y vuelta:
Te quiero y me quiero,
Te cuido y me cuido,
Te respeto y me respeto"

1302

Walter Riso

Analizador Sintáctico



Recibe código fuente en forma de tokens provenientes del Analizador Léxico y determina la estructura del programa

Los tokens se agrupan en estructuras sintácticas de la siguiente manera:

Sujeto Verbo Predicado

"El auto veloz corrió rápido en la carretera"

El: Artículo
auto: Sustantivo
veloz: Adjetivo
corrió: Verbo
rápido: Adjetivo
en la carretera: Predicado

Árbol Sintáctico

Estructura estándar basada en un apuntador asignado de forma dinámica. El árbol completo es una variable que apunta al nodo raíz; cada nodo en el árbol es un registro.

Los nodos y hojas se forman por expresiones o instrucciones, estas se componen de operandos, operadores y sus relaciones.

Tipos de Expresiones

Asignación
Expresiones con un :=

```
buaf := true;
```

Simple
Un solo objeto.

Aritmética
Operadores matemáticos

Ej. Considere las reglas como parte de la definición de expresiones

i) Cualquier identificador es una expresión.

ii) Cualquier literal es una expresión

iii) Si ID es un identificador y EXPR es una expresión, entonces ID1 := EXPR1 es una proposición.

iv) Si EXPR1 y EXPR2 son expresiones, también lo son:

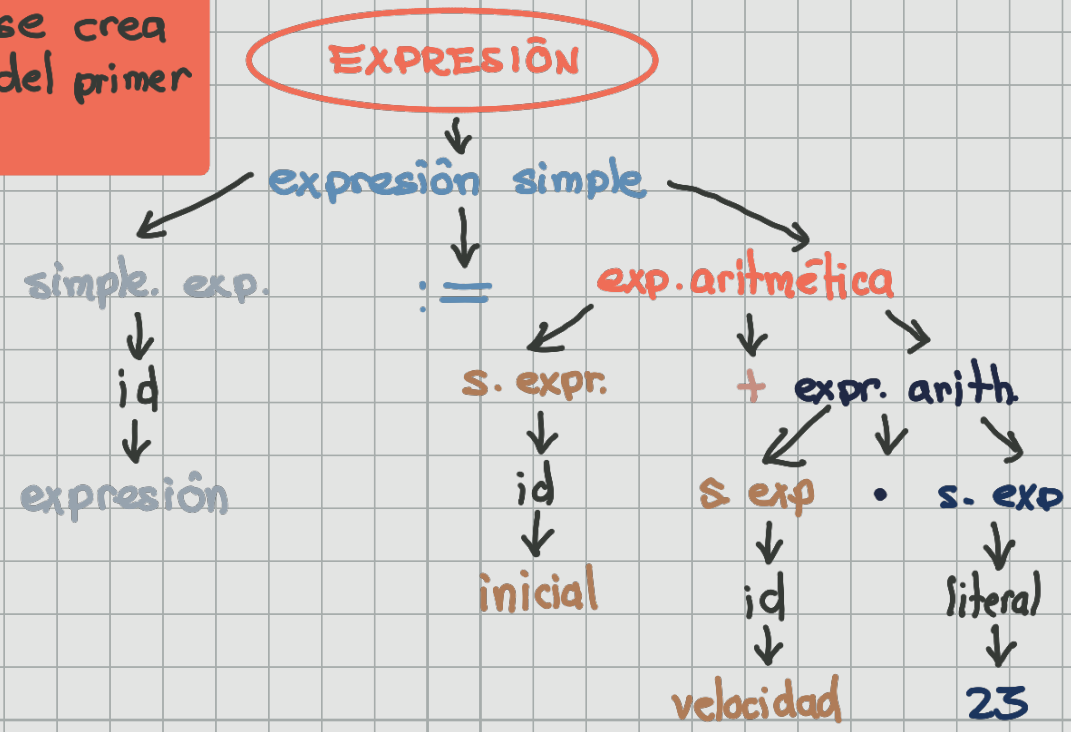
expr1 + expr2
expr1 - expr2
expr1 · expr2
expr1 / expr2
(expr1)

Construya el árbol sintáctico de:

posición := inicial + velocidad • 23

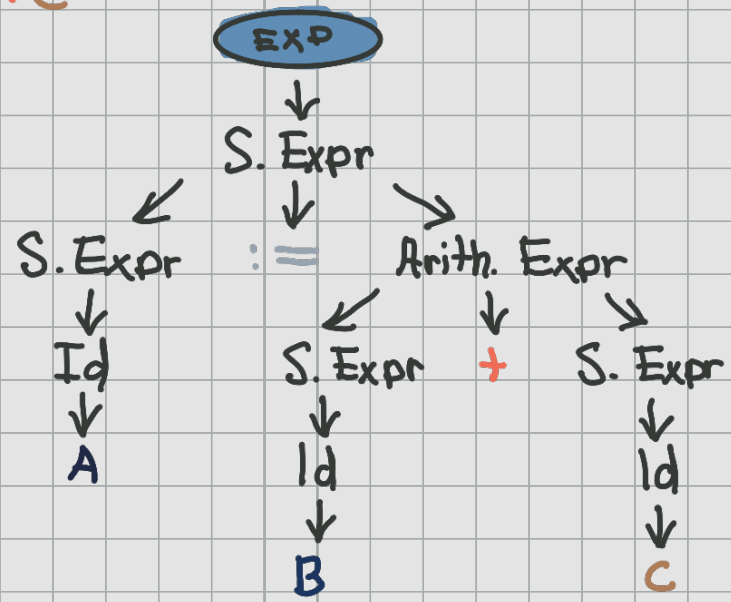
→ expresión de asignación — aritméticas

El árbol se crea a partir del primer operador

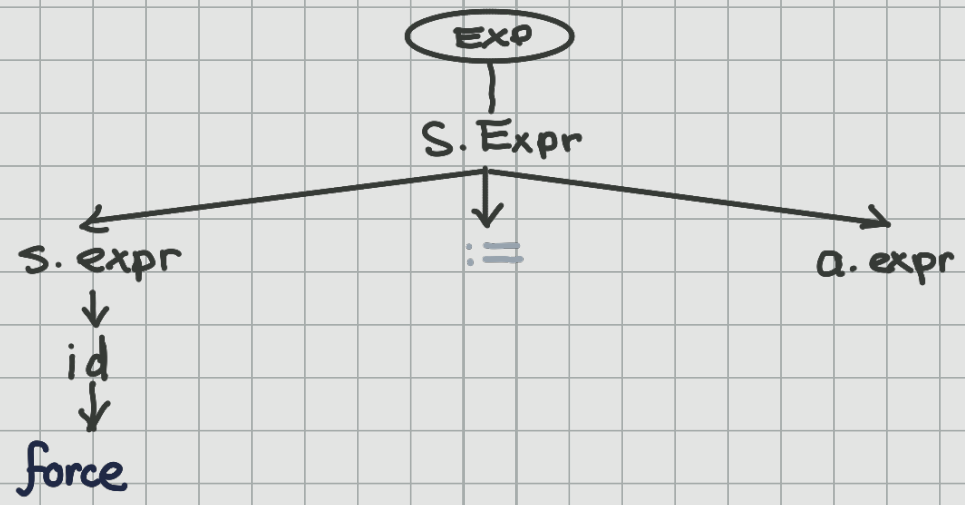


Ej. Genere el árbol sintáctico de:

a) A := B + C



b) Force := emotion + reason − pessimism

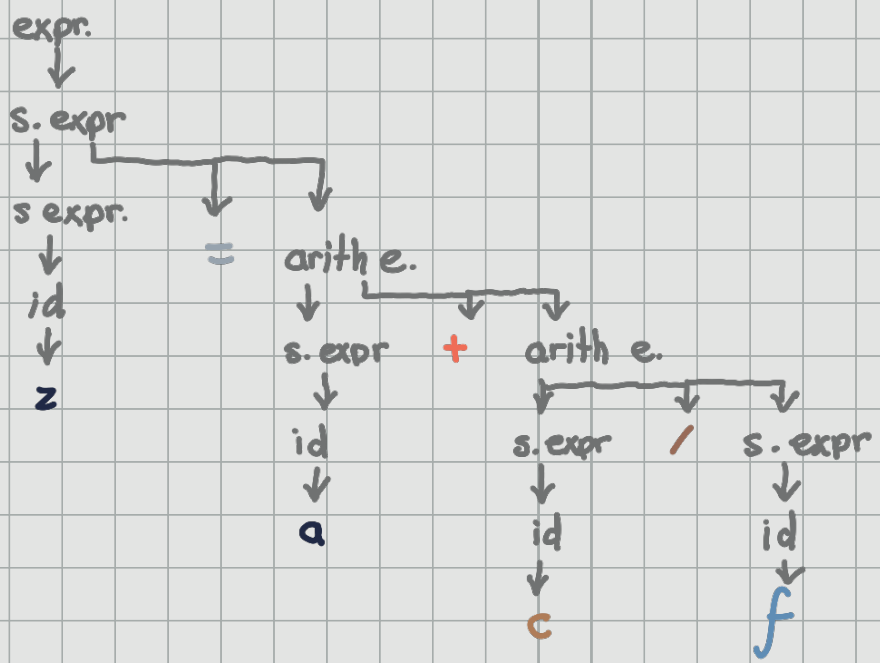


"Si crees que puedes, tienes razón; si crees que no puedes, también tienes razón.

Henry Ford

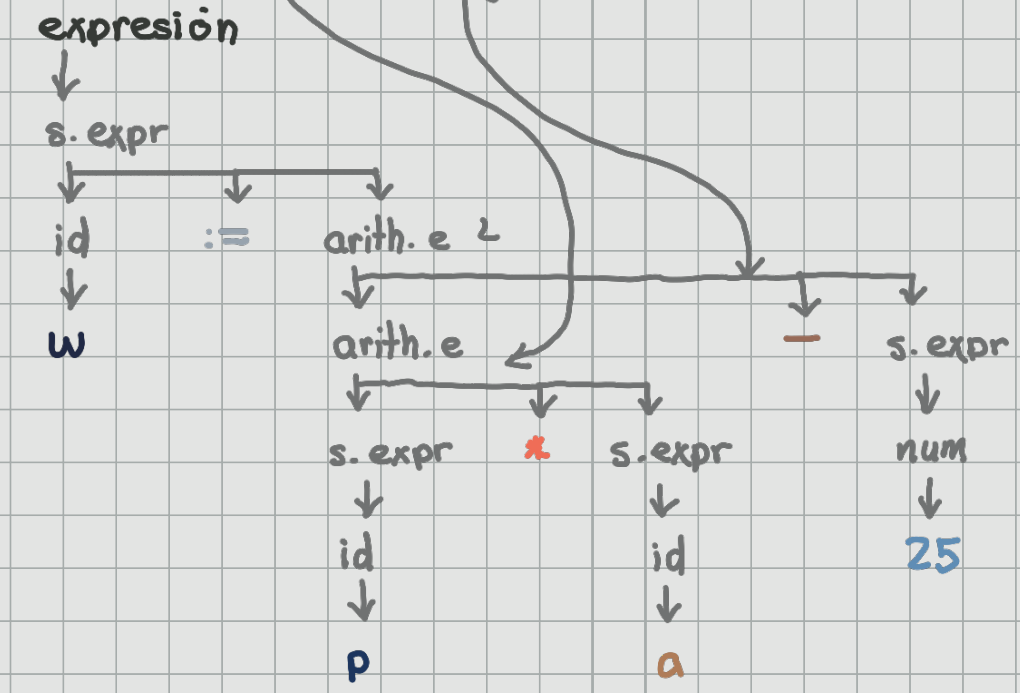
Ej. Cree el árbol de sintaxis. 18102

z = a + c / f

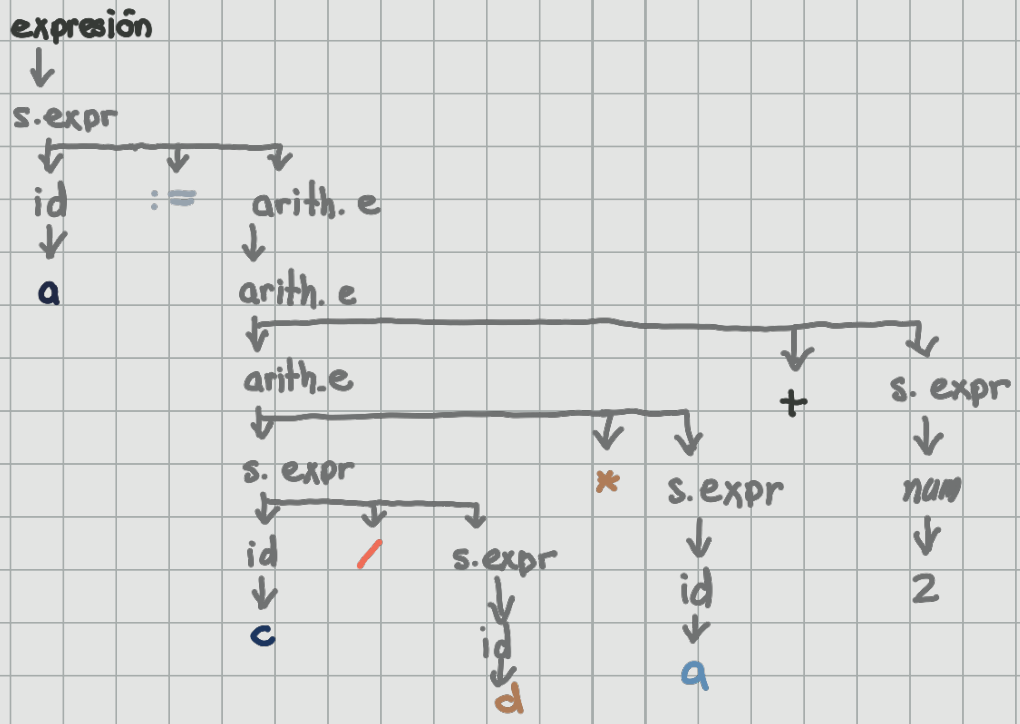


Ej. w := p * a - 25

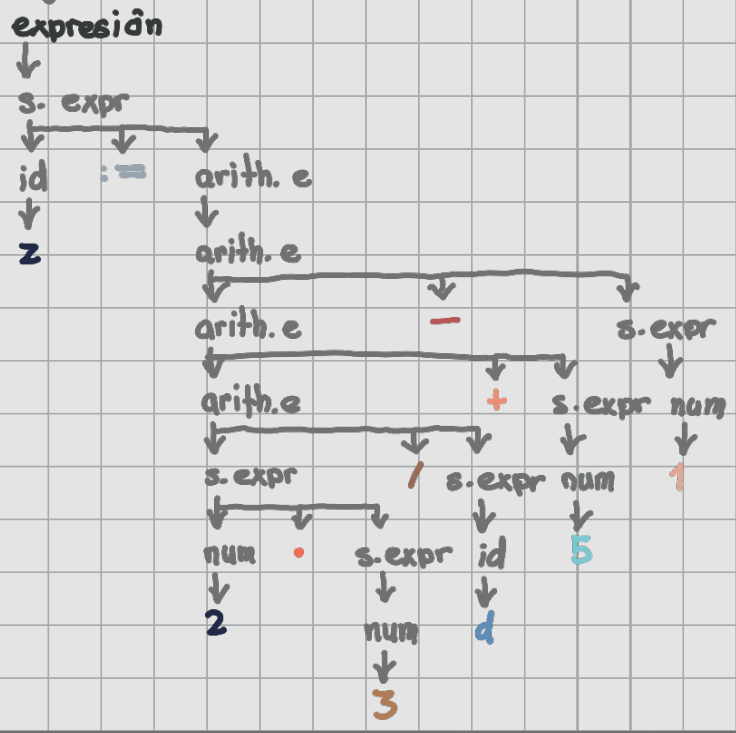
jerarquía de operadores der → izq



Ej. a := c/d * a + 2

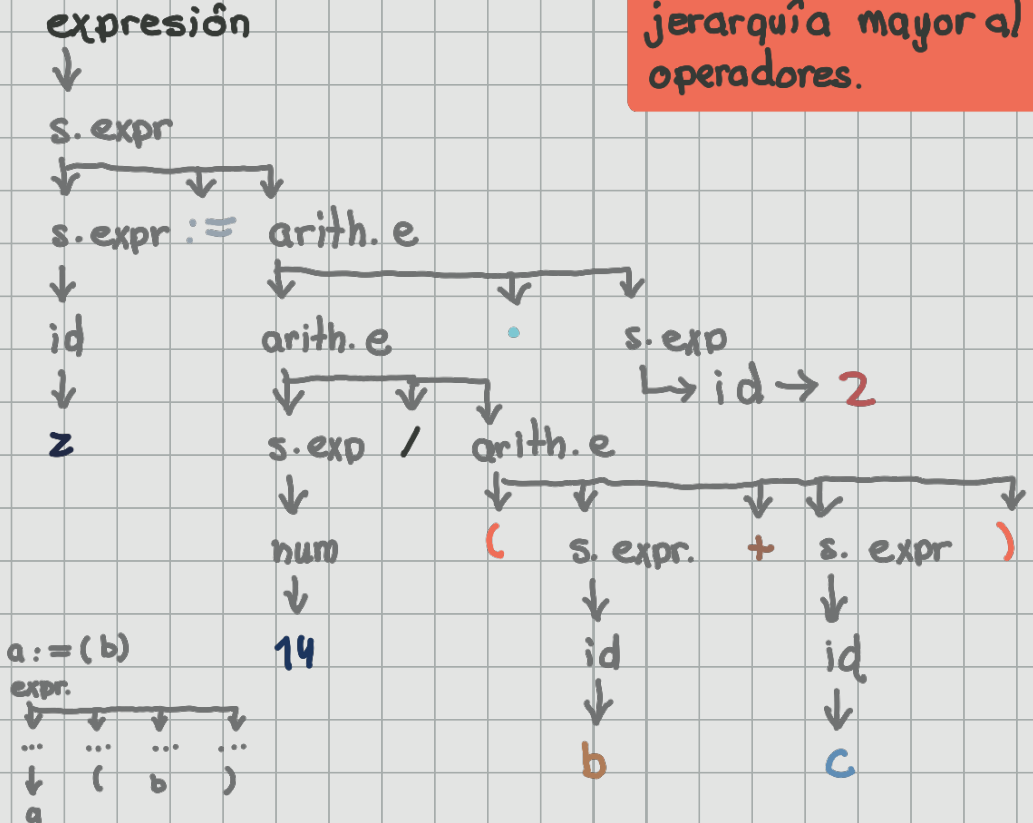


Ej. z := 2 · 3/d + 5 − 1



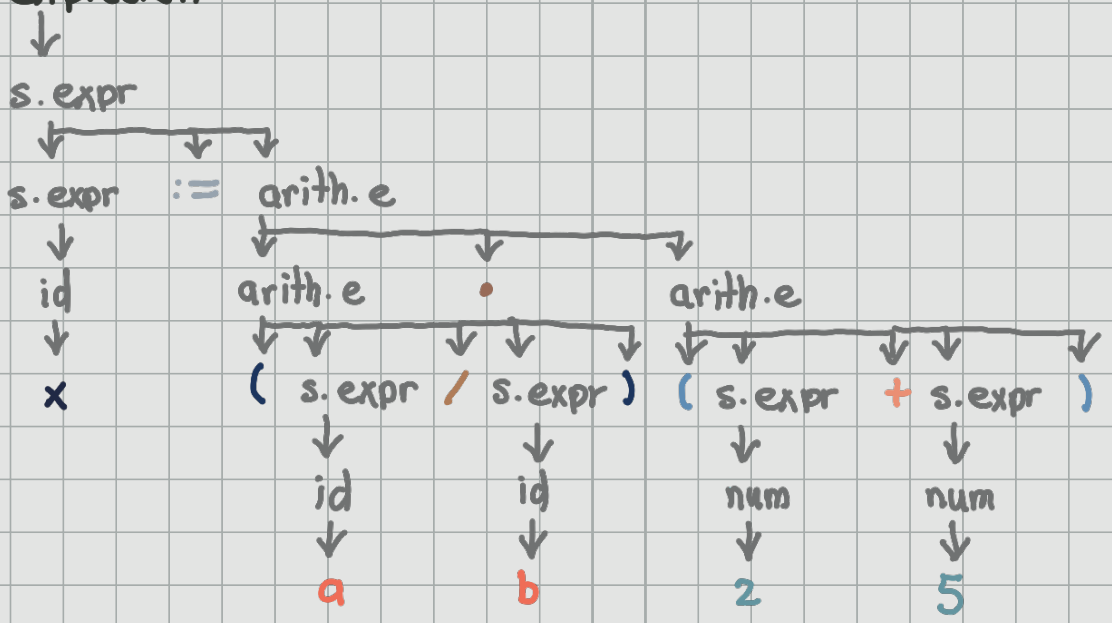
z := 14/(b+c) · 2

El paréntesis tiene una jerarquía mayor al resto de operadores.

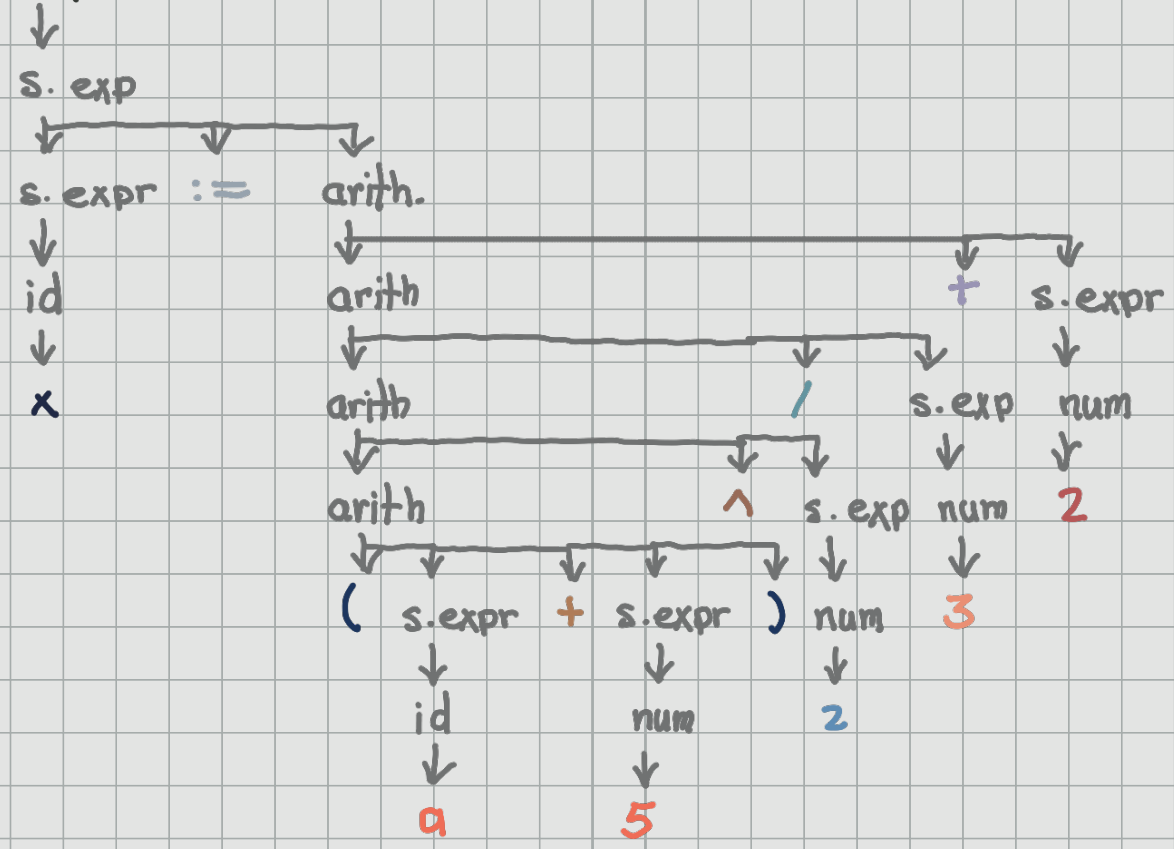


a := (b)

Ej. x := (a/b)·(2+5)



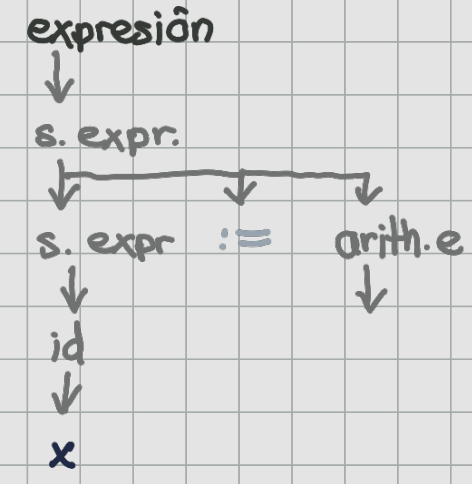
Ej. x := (a+5)^2/3+2



"La belleza está en los ojos de quien mira"

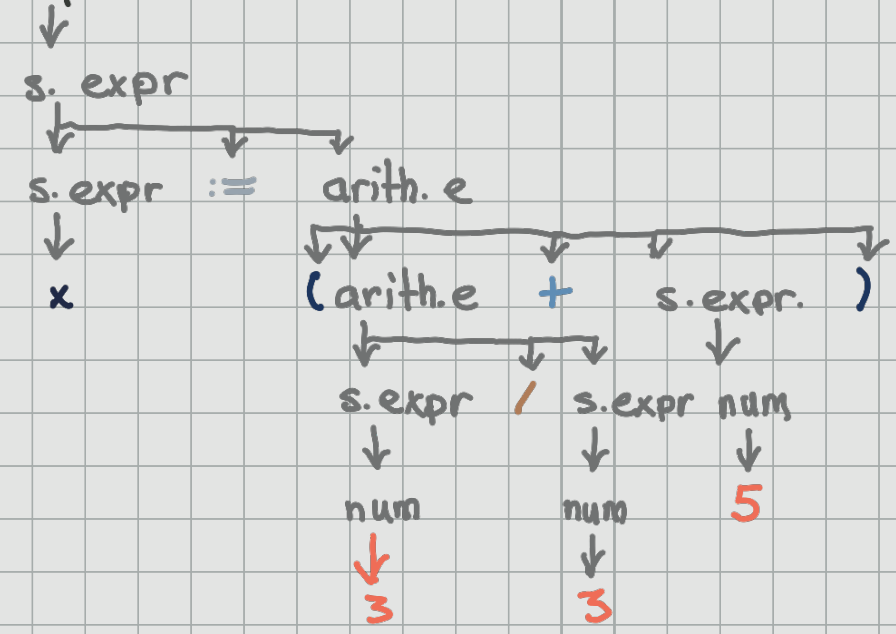
2012

x := (5 -4)/(6 +5)



dice el profe que no se puede xd

x := (3 /3+5)



Analizador Semántico

árbol sintáctico → **Análisis Semántico** → árbol semántico con anotaciones

La semántica de un programa es su significado, determina su comportamiento al ejecutarlo (semántica dinámica), pero casi todos los lenguajes de programación tienen características determinables antes de la ejecución (semántica estática) En Compilación se estudia la última.

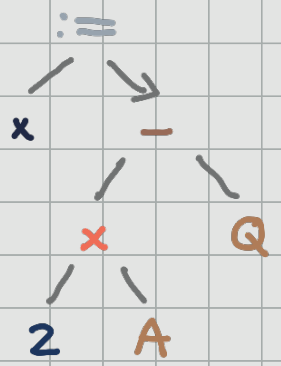
Al Árbol Sintáctico se le agregan anotaciones, tipos, para formar el Á. Semántico. Las anotaciones se llaman atributos.

Verificación del Programa	Generación de la representación intermedia
TAREAS DEL ANÁLISIS	

Árbol Abstracto

Omite las especificaciones de los operandos y operadores.
Aplica las mismas reglas que el árbol sintáctico.

$x := 2 * A - Q$ genera:

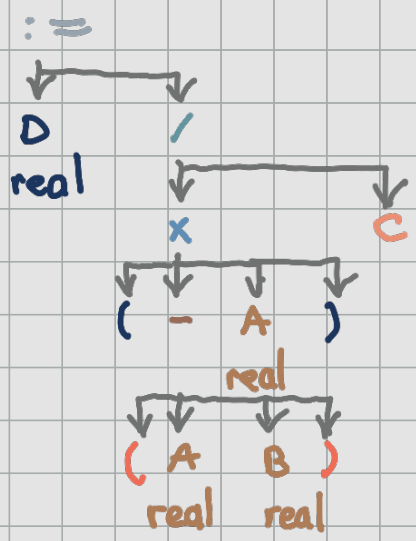


Árbol Semántico

Sólo operandos / operadores más las anotaciones.

Obtener árbol semántico abstracto de:

D := ((A − B) x A) / C



Optimización

Busca mejorar la forma en el que un programa usa los recursos del equipo de cómputo.

Tiempo de Ejecución y/o Tamaño de código

METAS

Nunca deberá afectar la ejecución del programa.

Tipos de Optimización

Local

Aplicada a un bloque simple aislado. Tienen una entrada y una salida.

Folding

Sustituye expresiones por sus resultados en lo posible.

Global

Aplicado a un programa o procedimiento completo.

Ej. Simplifique:

$w = 2+52-D+F+100$
$w = 57-D+F+100$
$w = 57+100-D+F$
$w = 157-D+F$

$p = 3-(5+6)-A*10+4$
$p = 3-(11)-A\cdot 10+4$
$p = -8-A\cdot 10+4$
$p = -4-A\cdot 10$

"Dios, concédeme serenidad para aceptar las cosas que no puedo cambiar; valor para cambiar aquellas que puedo; y sabiduría para reconocer la diferencia"

Propagación de Constante

25|02

Agrega valores constantes durante el cuerpo del código.

Ej. Considere las instrucciones
$$\begin{cases} x = 3; \\ \vdots \\ b = c + x; \end{cases}$$
suponga que entre instrucciones el valor de x es constante, por lo que el programa se podría modificar como:

```
x = 3;        [...]        b = c+3;
```

Ej. Optimizar: `Pi=3.1416; [...]; GI = Pi/180`

`GI = 3.1416/180` ——— propagación de constante

`GI = 0.017` ——— folding

Ej. Optimizar:

```
a=5;
x=2*a;
y=x+6;
```

$t = x \cdot y$

$\therefore$ $a = 5$

$x = 2 \cdot 5$ ——— propagación ...

$x = 10$ ——— folding

$y = 10 + 6$ ——— propag

$y = 16$ ——— folding

$t = 10 \cdot 16$ ——— propag

$t = 160$ ——— folding

Eliminación de Expresiones Comunes o Redundantes

Las expresiones con múltiples instancias se evalúan una única vez y se reutiliza el resultado. Siempre y cuando entre instrucciones que separan, el valor no se haya cambiado / asignado.

Ej. Considere: $\begin{cases} A = B \cdot C \\ \vdots \\ D = B \cdot C \end{cases}$

Suponga que la primera instrucción se ejecuta antes de la otra y que en el transcurso no hay cambio alguno.

Por lo tanto el código se escribe: $\begin{cases} T = B \cdot C \\ \vdots \\ A = T \\ D = T \end{cases}$

Simplificaciones Matemáticas

Realizadas a medida de lo posible. Se eliminan algunas expresiones

$x = y \wedge 2 \implies x = y \cdot y$

Ej: Simplificar

$a = x \wedge 2$; $b = 3$; $c = x$; $d = c \cdot c$; $e = b \cdot 2$; $f = a + d$; $g = e \cdot f$

Iteración 1:	It 2	it = 3	it = 4
a = x^2;	a = x·x	a = x·x	a = x·x
b = 3;	b = 3	b = 3	b = 3
c = x;	c = x	c = x	c = x
d = c·c;	d = x·x	d = a	d = a
e = b·2;	e = 3·2	e = 6	e = 6
f = a+d;	f = a+d	f = 2d	f = 2d
g = e·f;	g = 6·a+d	g = 6·2d	g = 12d

Ej. Optimizar:

$t_1 = 4-2$
$t_2 = t_1/2$
$t_3 = a \cdot t_2$
$t_4 = t_3 \cdot t_1$
$t_5 = t_4 + b$
$t_6 = t_3 \cdot t_1$
$t_7 = t_6 + b$

i) Folding

$t_1 = 2$
$t_2 = t_1/2$
$t_3 = a \cdot t_2$
$t_4 = t_3 \cdot t_1$
$t_5 = t_4 + b$
$t_6 = t_3 \cdot t_1$
$t_7 = t_6 + b$

ii) Propag.

$t_1 = 2$
$t_2 = 2/2$
$t_3 = a \cdot t_2$
$t_4 = t_3 \cdot t_1$
$t_5 = t_4 + b$
$t_6 = t_3 \cdot t_1$
$t_7 = t_6 + b$

iii) Redund/Propag

$t_1 = 2$
$t_2 = 1$
$t_3 = a \cdot 1$
$t_4 = t_3 \cdot 2$
$t_5 = t_4 + b$
$t_6 = t_4$
$t_7 = t_4 + b$

iv) Simp.

$t_1 = 2$
$t_2 = 1$
$t_3 = a$
$t_4 = 2 \cdot a$
$t_5 = t_4 + b$
$t_6 = t_4$
$t_7 = t_4 + b$

v) Simpl.

$t_1 = 2$
$t_2 = 1$
$t_3 = a$
$t_4 = 2 \cdot a$
$t_5 = t_4 + b$
$t_6 = t_4$
$t_7 = t_5$

Tenemos que ser conscientes de que todo proceso lleva su tiempo. No abandones algo que realmente quieras porque los resultados aún no se vean. Ninguna cosecha da frutas de la noche a la mañana.

Eliminación de Código Muerto

04|03

Omisión de código que nunca se ejecutará o es de operación insustancial.

Ej

```
b = FALSE;
if (a and b) then
    buaf();
```

como b es falsa, la condición nunca se cumplirá.

```
int global
void f() {
    int i;
    i = 1;
    global = 1;
    global = 2;
    return;
    global = 3;
}
```

no se usa

asignaciones innecesarias

ya no se ejecuta

Optimización Global

11|03

Antes de este proceso se debe crear el grafo de ejecución, el cual representa las rutas posibles de ejecución del programa y el optimizador.

La Optimización Global es costosa en tiempo de compilación.

Generador de Código Objeto

Transforma código intermedio en código de bajo nivel.

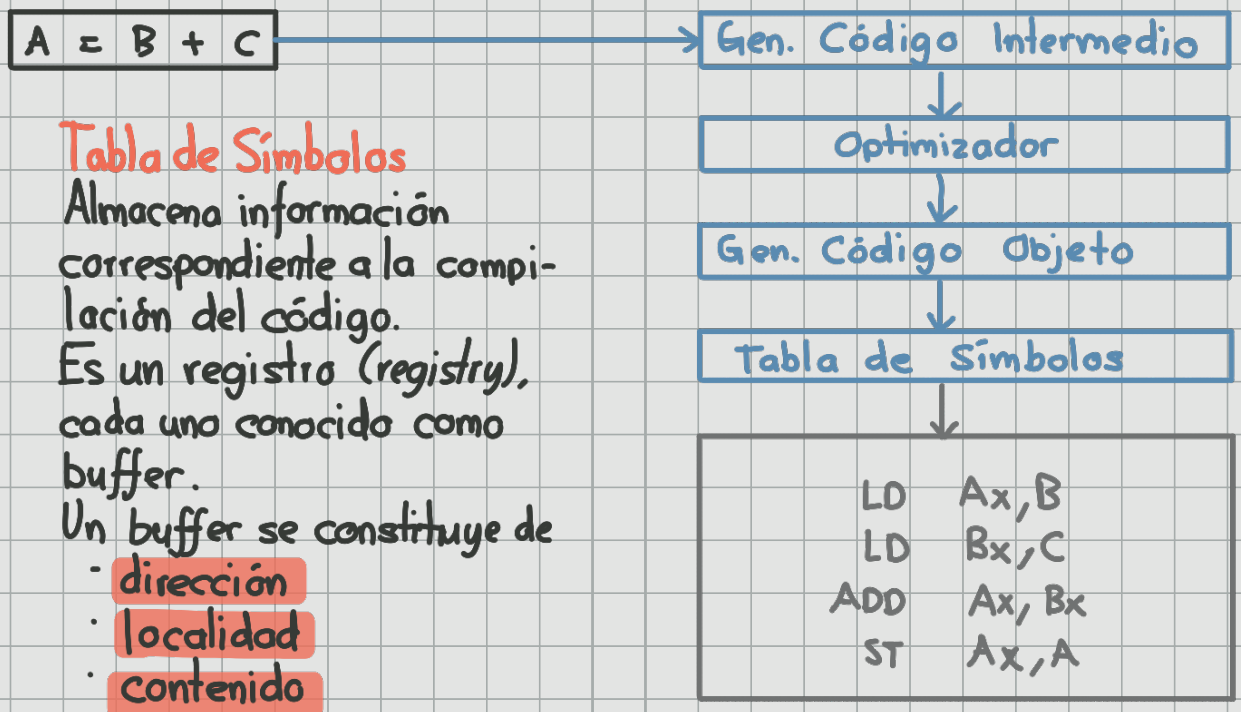


Tabla de Símbolos

Almacena información correspondiente a la compilación del código.
Es un registro (registry), cada uno conocido como buffer.
Un buffer se constituye de
- dirección
- localidad
- contenido

Es posible almacenar:

identificadores literales tipos

Puede realizar las operaciones:

Inserción Consulta Eliminación

Gestión de Errores

Detecta errores de tipo:

Léxico Sintáctico Semántico

Delegados a fases posteriores.

Recuperación de errores, donde para al encontrar al primer error y poder regresar a un contexto fiable.